SINH MÃ ĐỐI TƯỢNG TRÊN CƠ SỞ KẾT HỢP LẬP TRÌNH ĐỘNG VÀ SO TRÙNG CÂY

Nguyễn Chánh Thành - Phan Thị Tươi
Trường Đại học Bách Khoa – ĐHQG-HCM
(Bài nhận ngày 05 tháng 05 năm 2003)

TÓM TẮT: *Cũng như phân tích từ vựng và phân tích cú pháp sinh mã là vấn đề rất quan trọng của trình biên dịch. Bài báo này sẽ trình bày thuật toán chuyển đổi cây biểu thức sang mã đối tượng cho các máy tính thanh ghi. Thuật toán này tích hợp hai thuật toán so trùng cây nhanh từ trên xuống và lập trình động nhằm tối ưu mã đối tượng cho các lớp máy tính thanh ghi, và được thực thi với thời gian tuyến tính với dung lượng dữ liệu nhập vào.*

1. GIỚI THIỆU

Khi nghiên cứu bộ sinh mã đối tượng, chúng ta nhận thấy nếu áp dụng các kỹ thuật khác nhau để sinh mã và tối ưu mã đối tượng, thì kết quả sẽ khác biệt đáng kể giữa mã chưa tối ưu và tối ưu và giữa các phương pháp sinh mã đối tượng khác nhau.

Có nhiều thuật toán thực hiện việc tối ưu mã đối tượng như: sinh mã từ DAG (*directed acylic graph*)[1], sinh mã theo thuật toán lập trình động (*dynamic programming*)[1] và sinh mã theo thuật toán so trùng và viết lại cây (*tree matching-rewriting tree*)[1].

Bài báo này trình bày thuật toán tối ưu mã đối tượng dựa trên sự kết hợp của thuật toán lập trình động và so trùng và viết lại cây nhằm tạo mã đối tượng tối ưu hơn.

2. CÁC THUẬT TOÁN SINH MÃ ĐỐI TƯỢNG

Khi hiện thực sinh mã, dữ liệu là khối cơ bản và được biến đổi thành DAG. Giả thiết rằng dữ liệu đã được tối ưu trong giai đoạn sinh mã trung gian.

1. Thuật toán lập trình động

Thuật toán dựa trên nguyên tắc lập trình động để mở rộng khả năng của các máy tính, nó cho phép tối ưu mã đối tượng từ các cây biểu thức với thời gian tuyến tính. Thuật toán này được áp dụng cho một lớp khá lớn các máy tính thanh ghi có các tập chỉ thị phức tạp.

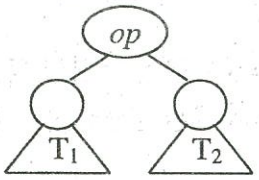
Thuật toán lập trình động được dùng để sinh mã cho các máy tính có r thanh ghi, được ký hiệu $R_0, R_1,..., R_{r-1}$ và các chỉ thị có dạng R_i:=E, trong đó E là một biểu thức chứa các toán tử, thanh ghi và vị trí nhớ. Nếu E bao gồm một hay nhiều thanh ghi thì R_i phải là một trong các thanh ghi đó.

Chúng ta xét trên một máy tính đơn giản chỉ có 3 chỉ thị có dạng R_i:=M, M:=R_i và R_i:=R_j và giả thiết rằng chi phí cho một chỉ thị là 1.

Thuật toán lập trình động phân hoạch bài toán sinh mã tối ưu cho một biểu thức thành các bài toán con để sinh mã tối ưu cho các biểu thức con.

i. Tính chất đánh giá tiếp giáp

Xét biểu thức E có dạng E_1 *op* E_2. Kết quả tối ưu cho E được tạo ra bằng cách tổng hợp các kết quả tối ưu của E_1 và E_2 theo một trong hai thứ tự, sau đó sinh mã cho toán tử *op*. Các bài toán sinh mã tối ưu cho E_1 và E_2 được giải quyết hoàn toàn tương tự.



Hình 1

Một kết quả tối ưu được sinh bởi thuật toán lập trình động. Tính chất quan trọng của nó là đánh giá biểu thức $E=E_1$ *op* E_2 theo kiểu *tiếp giáp* (*contiguous*). Xét biểu thức $E=E_1$ *op* E_2, và cây cú pháp T của biểu thức E gồm 2 cây con T_1 và T_2 tương ứng với E_1 và E_2 như ở hình 1.

P được gọi là "đánh giá tiếp giáp" một cây T nếu trước tiên nó đánh giá các cây con của T và lưu vào bộ nhớ, sau đó nó đánh giá phần còn lại của T theo thứ tự T_1, T_2 (hay T_2, T_1) và nút gốc. Trong mỗi trường hợp nó dùng các giá trị đã được tính trước từ bộ nhớ.

Ngược lại, đánh giá không giáp tiếp sẽ cho kết quả không tối ưu. Thí dụ P trước tiên đánh giá một phần của T_1 và lưu tạm trong thanh ghi (hay bộ nhớ) rồi sau đó đánh giá T_2 và tiếp tục đánh giá phần còn lại của T_1.

Nếu xét lớp máy tính thanh ghi thì một chương trình P cho trước trong ngôn ngữ máy đánh giá một cây biểu thức T thì chúng ta có thể tìm được một chương trình P' sao cho:

- P' có chi phí không cao hơn P
- P' dùng không nhiều thanh ghi hơn P
- P' đánh giá cây theo lối tiếp giáp

Kết quả này khẳng định mỗi cây biểu thức có thể được đánh giá tối ưu bằng một chương trình tiếp giáp. Tính chất này cho thấy với mọi cây biểu thức T, luôn tồn tại một chương trình tối ưu bao gồm các chương trình tối ưu cho các cây con của gốc; theo sau là một chỉ thị để đánh giá gốc. Điều này cho phép chúng ta dùng thuật toán lập trình động để sinh ra chương trình tối ưu cho cây T.

ii. Thuật toán lập trình động

Giả sử máy đích có r thanh ghi R_0, R_1,... R_{r-1} và mỗi nút n của cây T đều có một vectơ $C_1(n)$ (với $0<=i<=r$) để lưu chi phí:

- $C_1(n)$ là chi phí tối ưu khi tính cây con S có gốc tại n và lưu vào một thanh ghi. Giả thiết rằng có sẵn i thanh ghi để tính toán (với $0<=i<=r$). Chi phí bao gồm mọi thao tác tải và lưu thông tin để đánh giá S, được lưu trong $C_1(n)$.
- Chi phí cho việc thực hiện phép toán toán tử tại gốc của S được lưu trong $C_0(n)$.

Thuật toán này tiến hành theo trình tự sau:

Bước một: chúng ta tính vectơ chi phí $C_i(n)$ cho mỗi nút n của cây biểu thức T. Đánh giá tiếp giáp bảo đảm rằng một chương trình tối ưu S có thể được sinh ra bằng cách xét tổ hợp chương trình tối ưu của các cây con có gốc tại S. Ràng buộc này làm giảm đi số cây cần được xem xét.

Để tính $C_i(n)$ tại nút n, xét mỗi chỉ thị máy R:=E với biểu thức E so trùng với biểu thức con có gốc tại n. Xem xét các vectơ chi phí tại các hậu duệ tương ứng của n, để xác định chi phí đánh giá các toán hạng của E. Với các toán hạng của E là các thanh ghi, xét tất cả các thứ tự khả hữu để đánh giá cây con tương ứng của cây T vào các thanh ghi. Cho nút n, chúng ta thêm vào chi phí của chỉ thị R:=E đã được dùng để so trùng với nút n. Do đó giá trị $C_1(n)$ là chi phí nhỏ nhất trong tất cả các thứ tự khả hữu.

Vectơ chi phí cho toàn bộ cây T có thể tính từ dưới lên (*bottom-up*) với thời gian tỷ lệ tuyến tính với số nút trong T. Chúng ta lưu tại mỗi nút chỉ thị được dùng để thu được chi phí

tốt nhất cho $C_i(n)$ của nút n với mỗi giá trị của i. Chí phí nhỏ nhất trong vectơ của gốc của T cho biết chi phí nhỏ nhất để đánh giá T.

Bước hai: duyệt cây T bằng cách dùng các vectơ chi phí để xác định những cây con nào của T phải được cất vào bộ nhớ.

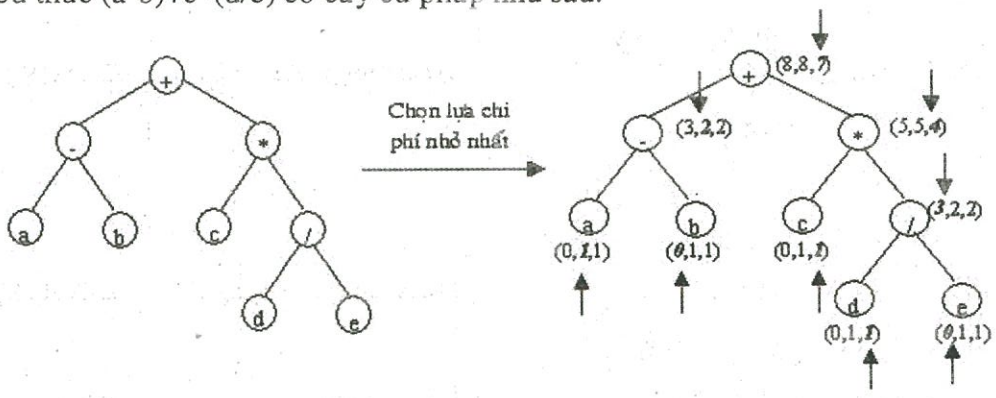
Bước ba: duyệt mỗi cây bằng cách dùng các vectơ chi phí và các chỉ thị đi kèm để sinh ra mã đích cuối cùng. Mã cho các cây con đã được cất vào các vị trí nhớ sẽ được sinh ra. Hai bước cuối này cũng được cài đặt với thời gian tuyến tính với kích thước của cây biểu thức.

Ví dụ: Xét một máy tính có 2 thanh ghi R_0, R_1 và các chỉ thị:

$R_i:=M_j$ $R_i:=R_i$ op R_j $R_i:=R_i$ op M_j $R_i:=R_j$ $M_i:=R_j$

Giả sử rằng các chỉ thị này đều có cùng một chi phí đơn vị, $R_i \in \{R_0, R_1\}$, M_j là vị trí bộ nhớ.

Xét biểu thức (a-b)+c*(d/e) có cây cú pháp như sau:



Và từ đó mã trung gian và mã đối tượng được sinh ra là:

Mã trung gian:		Mã đối tượng:	
	$R_0:=c$		MOV c, R_0
	$R_1:=d$		MOV d, R_i
	$R_1:=R_1/e$		DIV R_1, e
	$R_0:=R_0*R_1$		MUL R_1, R_0
	$R_1:=a$		MOV a, R_1
	$R_1:=R_1-b$		SUB R_1, b
	$R_1:=R_1+R_0$		ADD R_0, R_1

2. Thuật toán so trùng và viết lại cây

Dữ liệu nhập cho thuật toán so trùng và viết lại cây là cây ngữ nghĩa của máy đích. Thuật toán này dựa trên việc so trùng cấu trúc của cây ngữ nghĩa với từng quy tắc viết lại cây (rewriting tree) và từ đó thu giảm cấu trúc của cây, được dùng cho giai đoạn chọn chỉ thị tự động để sinh mã cho máy đích.

Mỗi quy tắc viết lại cây có dạng như sau:

replacement ⟵ template [{cost} = {action}]

Trong đó:

- replacement là một nút duy nhất dùng để thay thế trong quá trình thu giảm cây
- template là một quy tắc cho cấu trúc định sẵn được dùng trong quá trình so trùng cây
- cost là chi phí khi áp dụng quy tắc thu giảm cây
- action là một đoạn lệnh giống như trong lược đồ dịch hướng cú pháp

Một tập các quy tắc có dạng như trên được gọi là lược đồ dịch cây (tree-translation scheme)

Một số quy tắc được sử dụng trong việc so trùng như sau:

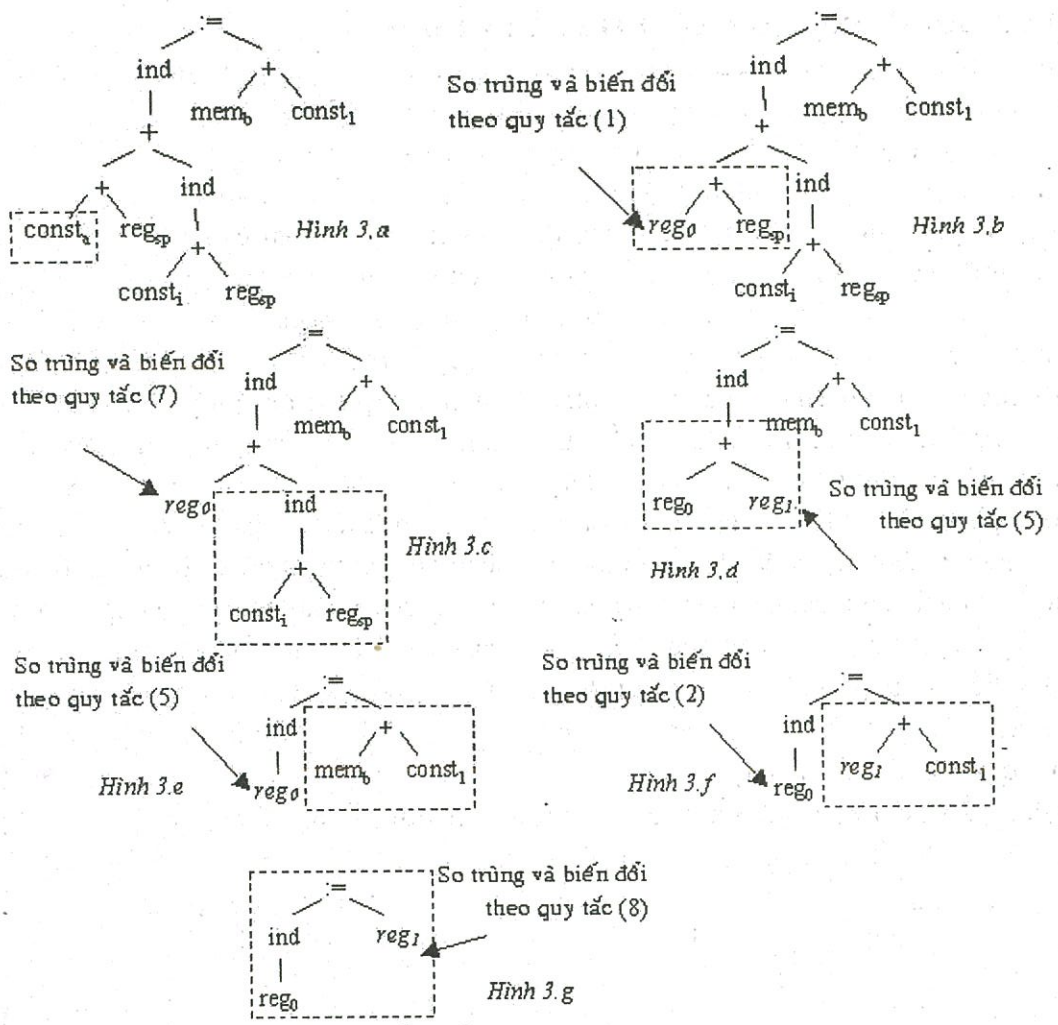
STT		Quy tắc	Chi phí	Lệnh phát sinh
(1)	reg_i	$\leftarrow const_c$	2	MOV #c, R_i
(2)	reg_i	$\leftarrow mem_a$	2	MOV a, R_i
(3)	mem	$\leftarrow$ (mem, reg_i)	2+cost(reg_i)	MOV R_i, a
(4)	mem	$\leftarrow$:= (ind, b); ind: reg_i	2+cost(reg_i)	MOV b, *R_i
(5)	reg_i	$\leftarrow$ + ($const_c$, reg_j)	2+cost(reg_j)	MOV c(R_j), R_i
(6)	reg_i	$\leftarrow$:= (reg_i, + ($const_c$, reg_j))	2+cost(reg_i)+cost(reg_j)	ADD c(R_j), R_i
(7)	reg_i	$\leftarrow$ + (reg_i, reg_j)	1+cost(reg_i)+cost(reg_j)	ADD R_j, R_i
(8)	reg_i	$\leftarrow$ + (reg_i, $const_1$)	1+cost(reg_i)	INC R_i

Mỗi quy tắc biểu diễn cho một phép toán được thực hiện bởi một các chỉ thị máy. Mỗi quy tắc tương ứng với một chỉ thị máy. Các nút lá của quy tắc là các thuộc tính có chỉ số giống như cây nhập.

Lược đồ dịch cây là một công cụ để chọn chỉ thị cho việc sinh mã. Trong quá trình hoạt động của lược đồ dịch cây với một cây nhập thì quy tắc viết lại cây được áp dụng cho các cây con của nó. Nếu so trùng được một quy tắc, cây con đã so trùng được thay bằng nút thay thế của quy tắc và hành động đi kèm với quy tắc sẽ được thực hiện. Nếu hành động chứa một dãy các chỉ thị máy, những chỉ thị này sẽ được xuất ra. Quá trình này được lập lại cho đến khi cây được thu gọn thành một nút, hoặc cho đến khi không có sự so trùng nào xảy ra. Dãy các chỉ thị máy sinh ra, chính là kết quả của lược đồ dịch cây trên cây dữ liệu đã cho.

Nếu nhiều quy tắc so trùng được với một cây con thì việc chọn lựa quy tắc phù hợp cần phải tuân theo nguyên tắc thu giảm để đạt được kết quả tối ưu nhất.

Ví dụ: Xét cây nhập chứa mã trung gian cho biểu thức a[i]:=b+1. Thực hiện so trùng các quy tắc với các thành phần của cây theo các hình dưới đây. Kết quả thu được trong hình 3.g dưới đây chỉ còn là một nút:



Theo trình tự thu giảm cây, kết quả thu được là:

MOV #a, R_0
ADD SP, R_0
ADD i(SP), R_0
MOV b, R_1
INC R_1
MOV R_1, *R_0

Nếu không có quy tắc so trùng nào được áp dụng thì quá trình sinh mã đối tượng bị thất bại, có thể một nút được thu giảm và viết lại bất tận – sinh ra một dãy vô hạn các chỉ thị di chuyển thanh ghi, chỉ thị tải hoặc lưu.

Chúng ta có thể mở rộng thuật toán so trùng cây để đạt được hiệu quả cao hơn:

- Bằng quy tắc so trùng từ trên xuống, trong đó mỗi quy tắc được biểu diễn bằng chuỗi đường dẫn đi từ gốc đến lá – vấn đề này sẽ được mô tả chi tiết trong thuật toán tiếp theo.
- Để sắp thứ tự các quy tắc được so trùng bằng thuật toán lập trình động ta cần xây dựng lược đồ dịch có bổ sung thêm thông tin về chi phí, liên kết mỗi quy tắc viết lại cây với chi phí của dãy chỉ thị được sinh ra.
- Để so trùng quy tắc bằng phân tích cú pháp, ta phải dùng lược đồ dịch hướng đến cú pháp bằng cách thay các quy tắc viết lại cây bằng các luật sinh của một văn phạm phi ngữ cảnh, trong đó vế phải là các biểu diễn tiền vị của các quy tắc chỉ thị.

III. SINH MÃ ĐỐI TƯỢNG BẰNG GIẢI THUẬT KẾT HỢP

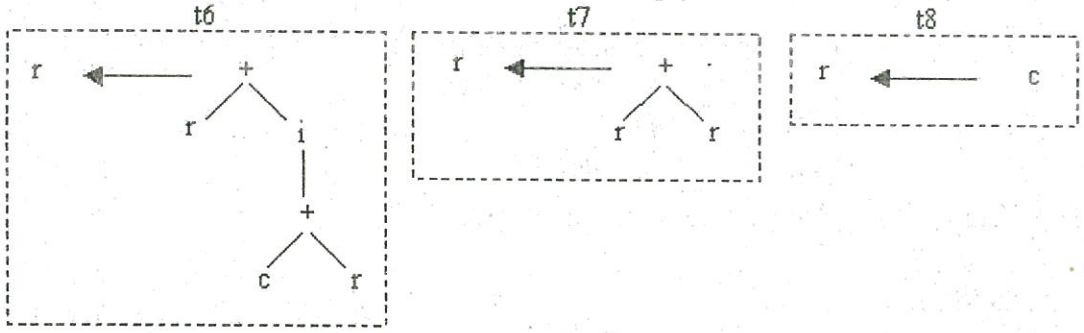
Quá trình viết lại cây có thể được cài đặt bằng cách cho thực thi việc so trùng quy tắc trong một lượt duyệt theo chiều sâu của cây nhập và thực hiện các thu giảm khi thăm viếng các nút lần cuối cùng. Nếu kết hợp với thuật toán lập trình động, chúng ta có thể chọn một dãy các so trùng tối ưu bằng cách dùng thông tin của chi phí kèm theo mỗi quy tắc. Chúng ta có thể hoãn lại quyết định về một so trùng cho đến khi chi phí của tất cả các khả năng đều đã được biết. Từ đó có thể sinh ra một chuỗi mã nhỏ hơn nhưng hiệu quả hơn, được xây dựng từ một lược đồ viết lại cây [4]. Thuật toán lập trình động còn giúp nhà thiết kế trong việc giải quyết những so trùng có xung đột hoặc đưa ra những quyết định về thứ tự đánh giá.

Một cách tiếp cận khác là dùng bộ phân tích cú pháp LR để thực hiện việc so trùng. Cây nhập được xử lý như một chuỗi biểu diễn tiền vị. Từ các luật sinh của lược đồ dịch, chúng ta có thể xây dựng bộ phân tích cú pháp LR.

Trong phần này, với ý tưởng của ngôn ngữ lập trình TWIG[3] mà nhóm nghiên cứu của A.V.Aho đã đưa ra, chúng tôi đề cập đến một phác thảo về giải thuật kết hợp, trong đó khai thác các ưu điểm của hai giải thuật lập trình động và so trùng và viết lại cây trong ngôn ngữ TWIG, mặc dù hiện tại ngôn ngữ TWIG chưa thực sự phát triển nhưng nó có nhiều tính năng mạnh.

Trước tiên, chúng ta sẽ tạo ra một tập các đường dẫn từ gốc đến các nút lá với quy tắc đánh dấu lần lược là 1, 2, ..., n cho các nhánh từ trái sang phải. Các đường dẫn này được sử dụng trong quá trình so trùng cây nhập.

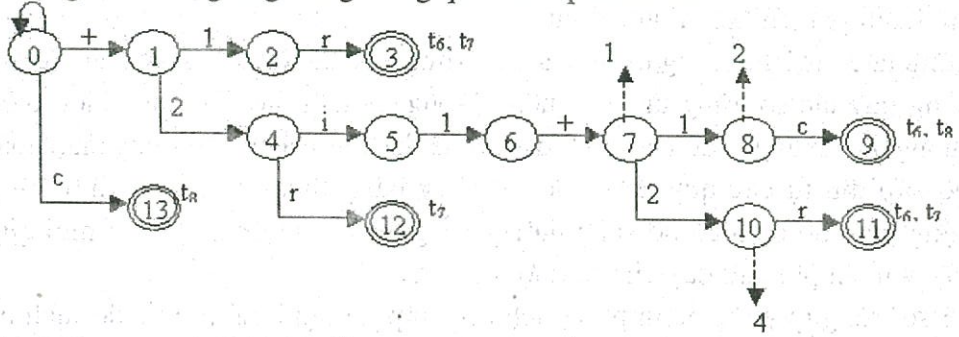
Ví dụ, giả sử chúng ta có 3 quy tắc (dựa theo các quy tắc 6,7, 8 nêu trên):



Với 3 quy tắc trên, sẽ có một tập các đường dẫn:

Với cây t6:	Với cây t7:	Với cây t8:
+1r	+1r	c
+2i1+1c	+2r	
+2i1+2r		

Lược đồ trạng thái tương ứng dùng trong quá trình phân tích:



Trong đó 0 là trạng thái khởi đầu, và các trạng thái kết thúc là 3, 9, 11, 12, 13. Tại trạng thái 7, nếu ký tự nhập kế tiếp không đúng thì sẽ gây lỗi và trở về trạng thái 1, tương tự cho trạng thái 8 là 2, cho trạng thái 10 là 4. Tại các trạng thái còn có lỗi sẽ quay về 0.

Đặt T = $\{1_i \leftarrow t_i\}$ với 1_i là nhãn và t_i là các quy tắc. Chúng ta có thể xây dựng một automat từ lược đồ trạng thái để chấp nhận các quy tắc trong T.

Gọi succ(δ, a) là trạng thái đến được từ trạng thái δ khi nhập vào ký tự a.

Automat tạo thông tin cho mỗi nút n của cây nhập dưới dạng mẫu tin bao gồm (parent, symbol, state).

Giả sử rằng các nút trong cây nhập và quy tắc có nhãn là ký hiệu như nhau. Giải thuật *visit* dưới đây sẽ duyệt cây theo chiều sâu và gán trạng thái cho mỗi nút, sau đó gọi post_process để xác định quy tắc.

```
Function visit(n)
        If (n là nút gốc) then
                n.state ← succc (0, n.symbol)
        Else
                n.state ← succc (succ(n.parent.state, k), n.symbol)
        End if
        // trong đó 0 là trạng thái khởi đầu, n là nút con thứ k là của n.parent
        For (mỗi nút con c của n) do
                Thực thi visit(c)
        End for
        Thực thi post_process(n)
End function
```

Với giải thuật *visit* trên, một chuỗi đường dẫn so trùng tại nút n của cây nhập nếu n.state là trạng thái kết thúc (được chấp nhận). Một nút n của cây nhập được xem là so trùng với quy tắc khi nút n so trùng với tất cả các chuỗi đường dẫn trong quy tắc đó.

Theo Hoffman và O'Donnell[5], việc xác định nút bắt đầu được so trùng thể hiện theo 2 phương pháp.

Phương pháp thứ nhất: với mỗi nút n có một tập biến đếm cho quy tắc tương ứng; Khi một chuỗi đường dẫn của quy tắc t_i được so trùng thì biến đếm tương ứng tại nút bắt đầu được so trùng sẽ được tăng lên 1 đơn vị. Khi giá trị của biến đếm này bằng với số chuỗi đường dẫn có trong quy tắc t_i thì ta nói n so trùng được với t_i.

Phương pháp thứ hai: mỗi nút trong cây nhập sẽ có một chuỗi bit $n.b_i$ (dùng để lưu thông tin cho việc so trùng cục bộ) liên kết với quy tắc t_i. Số bit trong $n.b_i$ bằng độ dài của đường dẫn dài nhất của t_i cộng thêm 1. Thứ tự các bit trong $n.b_i$ bắt đầu từ bit ở tận cùng bên phải, khi một chuỗi đường dẫn $s_{i,k}$ của t_i có độ dài 2j+1 thì bit thứ j của $n.b_i$ sẽ có trị 1, và điều này cũng có nghĩa là nếu bit thứ j của $n.b_i$ được đặt là 1 thì nút n (có độ sâu j) so trùng được với chuỗi đường dẫn con $s_{i,k}$ của quy tắc t_i. Để đạt tối ưu trong quá trình xử lý, chúng ta chỉ giữ lại những giá trị khác 0 trong chuỗi bit $n.b_i$ của nút n, như vậy giảm thiểu được một số lượng ô nhớ cần dùng thay vì phải lưu đầy đủ chuỗi bit $n.b_i$. Nếu với cây nhập có số lượng lớn các nút được xử lý bởi thuật toán này thì có thể tiết kiệm được một lượng ô nhớ khá lớn.

Với giải thuật *visit* trên thì việc xác định quy tắc sẽ được thực thi khi tất cả các nút con của n được duyệt qua và được gán trạng thái tương ứng trong lược đồ trạng thái nêu trên. Tại mỗi nút n thì một chuỗi bit $n.b_i$ mới được tính toán bởi việc dịch phải 1 bit (tương ứng với việc chia giá trị cho 2). Quá trình so trùng cây được thể hiện giải thuật *post_process*:

```
Function post_process
        For (mỗi b_i của nút n) do
                n.b_i ← 0
        End for
```

If (n.state là trạng thái kết thúc) then
Thực thi set_partial (n, n.state)
End if
For (mỗi quy tắc t_i) do
$n.b_i \leftarrow n.b_i$ *or* $(AND_{c \in C(n)} (c.b_i /2))$
End for
// với C(n) là tập các nút con c của nút n
Thực thi do_reduce (n)
End function
Và
Function set_partial (n, σ)
For (mỗi chuỗi đường dẫn $s_{i,k}$ của quy tắc t_i mà $s_{i,k}$ có độ dài 2j+1 và trạng thái σ được ghi nhận bởi $s_{i,k}$) do
$(n.b_i)_{[j]} \leftarrow 1$
End for
End function

Chương trình con *post_process* sẽ quản lý việc thu giảm cây. Chương trình con *set_partial* sẽ đặt bit thứ j của b_i tương ứng cho mỗi chuỗi đường dẫn $s_{i,k}$ (của t_i) ghi nhận được có độ dài 2j+1. Nhờ đó mà độ dài của đường dẫn so trùng ghi nhận được sẽ hiện hữu khi ở trạng thái kết thúc. Sau khi thực thi post_process xong thì t_i so trùng với các cây con của r nếu và chỉ nếu $r.b_i$ là lẻ, tức là bit tận cùng bên phải là 1.

Tiếp đến, cần tìm một bao phủ cho cây, sẽ được dùng trong việc thu giảm. Điều này có nghĩa là khi thành phần của cây t_i được ghi nhận thì việc thu giảm cho nhãn 1_i cần cho một thứ tự khi đi tìm bao phủ chứa so trùng của t_i. Giải thuật *do_reduce* sẽ cập nhật và thực hiện tính toán lập trình động (dựa trên chi phí của các quy tắc so trùng được) với nút n của cây nhập.

Function do_reduce(n)
For (với mỗi quy tắc t_I mà $(n.b_i)_{[0]}=1$) do
If ($cost(t_I, n) < n.cost[l_I]$) then
$n.cost[l_I] \leftarrow cost(t_I, n)$
$n.match[l_I] \leftarrow I$
if (n là nút gốc) then
$\delta \leftarrow succ\ (0, l_I)$
else
$\delta \leftarrow succ\ (succ(n.parent.state, k), l_I)$
// với n là nút con thứ k của n.parent
if (δ là trạng thái kết thúc) then
Thực thi set_partial (n, δ)
End if
End for
End function

Trong đó $cost(t_I, n)$ xác định bằng chi phí của quy tắc t_i so trùng tại nút n, chi phí này phụ thuộc vào chi phí so trùng tại các nút lá.

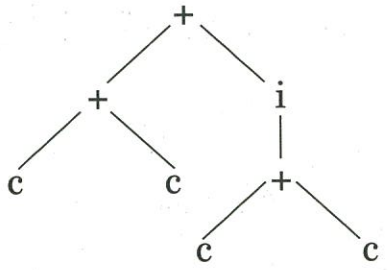
Với 3 quy tắc t6, t7, t8 nêu trên thì chi phí được tính tại một nút như sau:

$Cost(t_6, n)$ = 3+ chi phí tại các nút lá so trùng được

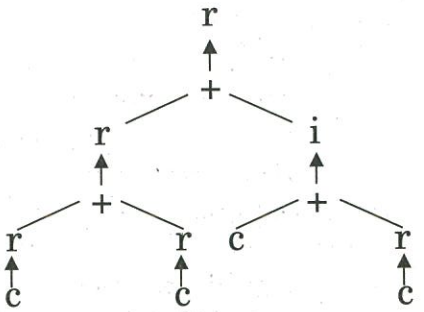
$Cost(t_7, n)$ = 1+ chi phí tại các nút lá so trùng được

$Cost(t_8, n)$ = 1

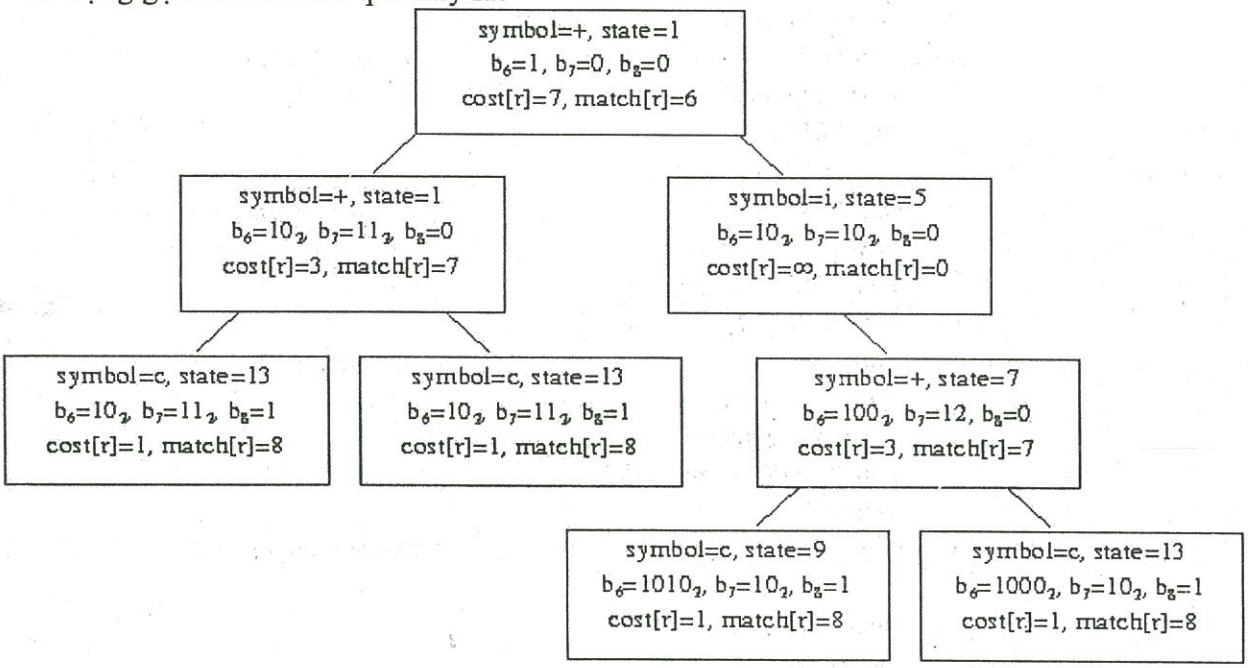
Với nhóm các giải thuật trên xét cây nguyên liệu sau:



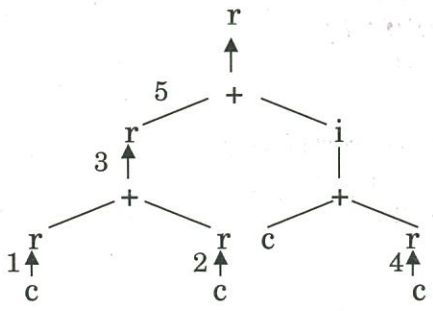
Khi tính toán xong, chúng ta sẽ có kết quả như sau:



Và dạng gợi nhớ của kết quả này là:



Giải thuật này tìm kiếm sự so trùng mà không bao gồm bất kỳ bao phủ nào. Thứ tự thu giảm cho cây nhập sẽ theo trình tự sau:



Giải thuật *do_rewrite*(n) sẽ thực thi quá trình thu giảm cây:

Function do_rewrite(n)

If (min{n.cost} = n.cost[i] và t.state là trạng thái kết thúc) then

Thực thi gencode(n)

//thực thi hành động tương ứng của quy tắc t_i

End if

Thực thi visit(n)

End function

Giải thuật *gencode*(n) thực thi việc sinh mã cho cây kết quả trên

```
Function gencode(n)
    Case n.match =1:                          //tương ứng quy tắc 1
        cost=2;
        gencode=getreg();
        emit('MOV', l1, regnode);
        return (regnode);
    Case n.match=2                            //tương ứng quy tắc 2
        cost=2;
        regnode=getreg();
        emit('MOV', l1, regnode);
        return (regnode);
    Case n.match=3                            //tương ứng quy tắc 3
        cost=2+l1.cost;
        emit('MOV', l2, l1);
        return (NULL);
    Case n.match=4                            //tương ứng quy tắc 4
        cost=2+l1.cost;
        emit('MOV', l2, l1);
        return (NULL);
    Case n.match=5                            //tương ứng quy tắc 5
        cost=2+l1.cost;
        regnode=getreg();
        emit('MOV', n, regnode);
        return (regnode);
    Case n.match=6                            //tương ứng quy tắc 6
        cost=2+l1.cost +l2.cost
        emit(op(n), l2, l1)
        return (l1)
    Case n.match=7: Case n.match=8            //tương ứng quy tắc 7,8
      If (value(l2)=1) then
        cost=1+l1.cost
        emit('INC', l1)
        return (l1)
      else
        cost=1+l1.cost+l2.cost
        emit(op(n), l2, l1)
        return (l1)
      end if
End function
```

KẾT LUẬN

Với việc hiện thực một giải thuật sinh mã tự động dựa trên ý tưởng của ngôn ngữ lập trình TWIG mà nhóm nghiên cứu của A.V.Aho[3] đã đưa ra, việc tối ưu mã đối tượng ngày càng trở thành vấn đề quan trọng được sự quan tâm của nhiều nhóm nghiên cứu. Trong phạm vi bài báo này, chúng tôi muốn làm sáng tỏ ý tưởng kết hợp giữa 2 giải thuật khá phổ biến mà nhiều nhóm nghiên cứu đã từng thực hiện bằng những giải thuật ở trên và đã thu được những kết quả nhất định. Tuy nhiên cũng có một số hướng nghiên cứu khác với việc

phát triển kỹ thuật viết lại cây dựa vào cây quyết định [6] đó cũng là giải pháp để tối ưu mã đối tượng.

CODE GENERATION BASED ON COMBINATION OF DYNAMIC PROGRAMMING AND TREE-PATTERN MATCHING ALGORITHMS

Nguyen Chanh Thanh - Phan Thi Tuoi

ABSTRACT: *After compiler-component generators, such as lexical analyzer and parser a code generator is very important. This paper presents algorithm, which transforms expression trees into code for register machines. This algorithm combines a fast top-down tree-pattern matching algorithm with dynamic programming, which produces optimal code for any machine in this class; and this algorithm runs in time linearly proportionals to the size of the input data.*

Additional Key Words and Phrases: Code generation, code generator-generator, code optimization, dynamic programming, pattern matching.

TÀI LIỆU THAM KHẢO

[1] Aho, A.V., Sethi, R.and Ullman, J.D.(1986) Compiler-Principles, Techniques, and Tools, Addison-Wesley, Reading, MA.

[2] Balachandran, A., Dhamdhere, D.M.and Biswas, S.(1990) Eficient retargetable code generation using bottom-up tree pattern matching. Computer Languages, 3, 127-40.

[3] Aho, A.V., Ganapathi, M.and Tjiang, S.W.K.(1989) Code generation using tree matching and dynamic programming. ACMTransactions on Programming Languages and System, 1, 159-75.

[4] Kwang, K.-W.and Choe, K.=M.(1995) On the automatic generation of instruction selector using bottom-up tree pattern matching. Tech.Rept. CS-TR-95-93, Dept.of Computer Science, KAIST.

[5] Hoffman, C. W., O'Donnell, M. J. Pattern matching in trees. J. ACM 29, 1 (1982), 68-95.

[6] Kyung-Woo Kang, Kwang-Moo Choe, Min-Soo Jung, Heung-Chul Shin An efficient bottom-up tree pattern matching that performs dynamic programming for code generation. Journal of Programming Languages 5, 1997, 189–199